All-Pairs Shortest Paths (Ch. 25)
The all-pairs shortest path problem (APSP)
input: a directed graph \( G = (V, E) \) with edge weights
goal: find a minimum weight (shortest) path between every pair of vertices in \( V \)
Can we do this with algorithms we've already seen?

Solution 1: run Dijkstra's algorithm \( V \) times, once with each \( v \in V \) as the source node (requires no negative-weight edges in \( E \))
- If \( G \) is dense with an array implementation of \( Q \)
  \( O(V^2) = O(V^3) \) time
- If \( G \) is sparse with a binary heap implementation of \( Q \)
  \( O(V((V + E) \log V)) = O(V^2 \log V + VE \log V) \) time

Solution 2: run the Bellman-Ford algorithm \( V \) times (negative edge weights allowed), once from each vertex.
- \( O(V^2 E) \), which on a dense graph is \( O(V^4) \)

Solution 3: use an algorithm designed for the APSP problem.
E.g., Floyd's Algorithm

Warshall's Transitive Closure Algorithm

Input: Adjacency matrix \( A \) of \( G \) as matrix of 1s and 0s
Output: Transitive Closure or reachability matrix \( R^{(n)} \) of \( G \)
Assumes vertices are numbered 1 to \( |V| \), \( |V| = n \) and there are no edge weights. Finds a series of boolean matrices \( R^{(0)}, \ldots, R^{(n)} \)

Solution for \( R^{(n)} \):
- Define \( r_{ij}^{(k)} \) as the element in the \( i \)th row and \( j \)th column to be 1 iff there is a path between vertices \( i \) and \( j \) using only vertices numbered \( \leq k \).
- \( R^{(0)} = A \), original adjacency matrix (only 1's are direct edges)
- \( R^{(k)} \) the matrix we want to compute
- \( R^{(0)} \)'s elements are: \( R^{(0)}[i,j] = r_{ij}^{(0)} = r_{ij}^{(k-1)} \lor r_{ik}^{(k-1)} \land r_{kj}^{(k-1)} \)

Warshall's Algorithm

\begin{align*}
\text{Warshall} (A[1 \ldots n,1 \ldots n]) \\
1. & n = \text{rows}[A] \\
2. & R^{(0)} = A \\
3. & \text{for } k = 1 \text{ to } n \text{ do} \\
4. & \quad \text{for } i = 1 \text{ to } n \text{ do} \\
5. & \quad \quad R^{(k)}[i,j] = r_{ij}^{(k)} = r_{ij}^{(k-1)} \lor r_{ik}^{(k-1)} \land r_{kj}^{(k-1)} \\
6. & \quad \text{endfor} \\
7. & \text{return } R^{(n)}
\end{align*}

If an element \( r_{ij} \) is 1 in \( R^{(k)} \), it remains 1 in \( R^{(k)} \).
If an element \( r_{ij} \) is 0 in \( R^{(k)} \), it becomes a 1 iff the element in row \( i \) and column \( k \) and the element in column \( j \), row \( k \) are both 1's in \( R^{(k-1)} \).

Warshall's Algorithm

Matrix \( R^{(0)} \) contains the nodes reachable in one hop
For \( R^{(1)} \), there is a 1 in row 3, col 1 and col 2, row 1, so put a 1 in position 3,2. Also, there is a 1 in row 3, col 1, and col 3, row 1 so put a 1 in position 3,3

Matrix \( R^{(1)} \) contains the nodes reachable in one hop or on paths that go through vertex 1.
For \( R^{(2)} \), there is no change because 1 can get to 3 through 2 but there is already a direct path between 1 and 3.
Warshall's Algorithm

Matrix $R^{(2)}$ contains the nodes reachable in one hop or on paths that go through vertices 1 or 2.

For $R^{(3)}$, there is a 1 in row 1, col 3 and col 1, row 3, so put a 1 in position 1,1. Also, there is a 1 in row 2, col 3 and col 1, row 3, so put a 1 in position 2,1. Also, there is a 1 in row 2, col 3 and col 2, row 3, so put a 1 in position 2,2.

Warshall's Algorithm:

1. $n = \text{rows}[A]$
2. $R^{(0)} = A$
3. for $k = 1$ to $n$ do
4. for $i = 1$ to $n$ do
5. for $j = 1$ to $n$ do
6. $R^{(k)} = R^{(k-1)} \land R_{k+1}^{(k-1)}$
7. return $R^{(n)}$

Time efficiency? Space efficiency?
Floyd's APSP Algorithm

Input: Adjacency matrix A
Output: Shortest path matrix D
and predecessor matrix Π

Observation: When G contains no negative-weight cycles, all shortest paths consist of at most n – 1 edges.

Relies on the Optimal Substructure Property:
All sub-paths of a shortest path are shortest paths.

Solution for D:

\[ D^{(k)} \] is as the minimum weight of any path from vertex i to vertex j, such that all intermediate vertices are in \(1, 2, 3, ..., k\)

\[ D^{(0)} = A, \text{ original adjacency matrix (only paths are single edges)} \]

\[ D^{(n)} \text{ the matrix we want to compute} \]

\[ D^{(k)} \text{'s elements are: } D^{(k)}[i, j] = d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}) \]

Assumes vertices are numbered 1 to |V|

Recursive Solution for D

\[ \pi^{(k)}_{ij} \text{ } \pi^{(k)}_{ij} \text{ is predecessor of } j \text{ on some shortest path from } i \]

Initial Matrix of Path Lengths

Use adjacency matrix A for G = (V, E):

\[ A[i, j] = d_{ij} = \begin{cases} \w(i, j) & \text{if } (i, j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases} \]

Initial Matrix of Predecessors

Use adjacency matrix Π to keep track of predecessors:

\[ \pi^{(0)}_{ij} = \begin{cases} i & \text{if } i \neq j \text{ and } w(i, j) < \infty \\ \emptyset & \text{if } i = j \text{ or } w(i, j) = \infty \end{cases} \]

Floyd's APSP Algorithm

Floyd-Warshall-APSP(A)

1. n = rows[A]
2. \( D^{(0)} = A \)
3. for k = 1 to n
4. for i = 1 to n
5. for j = 1 to n
6. if \( d^{(k-1)}_{ij} > d^{(k-1)}_{ik} + d^{(k-1)}_{kj} \)
7. \( d^{(k)}_{ij} = d^{(k-1)}_{ik} + d^{(k-1)}_{kj} \)
8. \( \pi^{(k)}_{ij} = \pi^{(k-1)}_{kj} \)
9. else \( \pi^{(k)}_{ij} = \pi^{(k-1)}_{ik} \)
10. return \( D^{(n)} \)

Operation of F-APSP Algorithm

\[ D^{(0)} = A = \begin{bmatrix} 0 & 4 & \infty \\ 6 & 0 & 2 \\ 3 & 5 & 0 \end{bmatrix} \]

\[ \Pi^{(0)} = \begin{bmatrix} \emptyset & 1 & 1 \\ 2 & \emptyset & 2 \\ 3 & \emptyset & \emptyset \end{bmatrix} \]
Operation of F-APSP Algorithm

\[ D^{(0)} = A = \begin{bmatrix} 0 & 3 & 8 & 4 & \infty \\ \infty & 0 & 7 & 1 & \infty \\ \infty & 4 & 0 & \infty & \infty \\ \infty & \infty & \infty & 0 & 6 \\ 2 & \infty & -5 & \infty & \infty \end{bmatrix} \]

\[ D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & -4 & \infty \\ \infty & 0 & 7 & 1 & \infty \\ \infty & 4 & 0 & \infty & \infty \\ \infty & \infty & \infty & 0 & 6 \\ 2 & \infty & -5 & \infty & \infty \end{bmatrix} \]

Operation of F-APSP Algorithm

\[ D^{(5)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 2 & 7 & 0 \end{bmatrix} \]

Operation of F-APSP Algorithm

\[ D^{(3)} = \begin{bmatrix} \emptyset & 1 & 2 \\ 2 & \emptyset & 2 \\ 3 & 1 & \emptyset \end{bmatrix} \]

Operation of F-APSP Algorithm

\[ D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 2 & 7 & 0 \end{bmatrix} \]

Printing the shortest path

Print-APSP(p, i, j) if p is predecessor matrix
1. if i = j
2. print i
3. else if p[i][j] = NIL
4. print “no path from * + i + * to * + j + * exists”
5. else
6. Print-APSP(p, i, p[i][j])
7. print j

F-APSP Algorithm

Print-path(p, i, j) // p is predecessor matrix
1. if i = j
2. print i
3. else if p[i][j] = NIL
4. print “no path from * + i + * to * + j + * exists”
5. else
6. Print-APSP(p, i, p[i][j])
7. print j
F-APSP Algorithm

\[ D^{(5)} = \begin{pmatrix}
0 & 3 & 8 & 4 & 2 \\
\infty & 0 & \infty & 7 & 1 \\
\infty & 4 & 0 & 11 & 5 \\
\infty & \infty & \infty & 6 & \infty \\
2 & -1 & -5 & 2 & 0
\end{pmatrix} \]

Running Time of Floyd's-APSP

Lines 3 – 6: \( |V|^3 \) time for triply-nested for loops

Overall running time = \( \theta(V^3) \)

The code is tight, with no elaborate data structures and so the constant hidden in the \( \theta \)-notation is small.